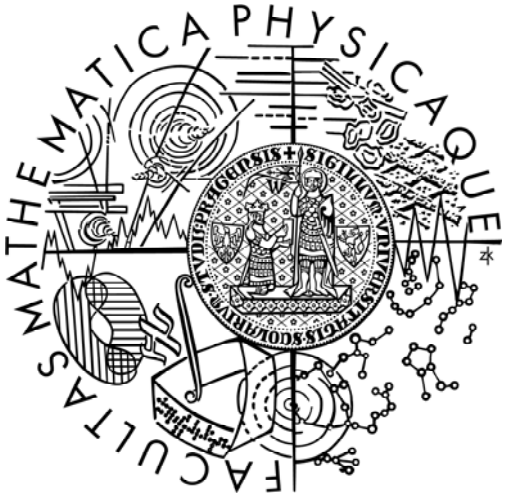
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Robert Goldwein

Processing of secondary structures in nucleic acids

Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. RNDr. Iveta Mrázová, CSc., KTIML

Studijní program: Informatika, Obecná informatika

2009

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 1. 6. 2009 Robert Goldwein

Table of Contents

Název práce: Processing of secondary structures in nucleic acids
Autor: Robert Goldwein
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: doc. RNDr. Iveta Mrázová, CSc., KTIML
e-mail vedoucího: Iveta.Mrazova@mff.cuni.cz

Abstrakt: Předložená práce se zabývá studiem základních metod v *bioinformatice* – v novém a perspektivním odvětví informatiky. Vysvětluje vlastní pojmem bioinformatika, seznamuje s nutnými biologickými základy (molekuly DNA, RNA, proteiny, centrální dogma molekulární biologie) a se základními bioinformatickými pojmy (biologická sekvence a její primární a sekunární struktura). Dále demonstruje implementaci základních bioinformatických algoritmů a jejich použití na reálných datech (na viru slintavky a kulhavky) – vyhledávání motivů, nejdelší společná podsekvence a alignment sekvencí. Práce také seznamuje s vyššími strukturami, především pak se sekundární strukturou RNA.

Klíčová slova: bioinformatika, zarovnání sekvencí, analýza sekvencí, sekundární struktura, dynamické programování

Title: Processing of secondary structures in nucleic acids
Author: Robert Goldwein
Department: Department of Software Engineering
Supervisor: doc. RNDr. Iveta Mrázová, CSc., KTIML
Supervisor's e-mail: Iveta.Mrazova@mff.cuni.cz

Abstract: This work explores and studies basic methods of *bioinformatics* – new and perspective branch of computer science. Introduces the term Bioinformatics, familiarizes with necessary biological background (DNA and RNA molecules, proteins, central dogma of molecular biology) and also with basic bioinformatics terms (biological sequence, primary and secondary structure). It also demonstrates the implementation of basic bioinformatics algorithms and their use with real data (on Foot-and-mouth disease virus) – motif finding, longest common subsequence and sequence alignment. This work also introduces higher structures of biological sequences, primarily with secondary structure of RNA molecule.

Keywords: bioinformatics, sequence alignment, sequence analysis, secondary structure, dynamic programming

Overview

The goal of this work is to explore and study basic methods of bioinformatics and familiarize the reader with this new and perspective branch of computer science.

First chapter introduces the term Bioinformatics, biological sequences and gives overview of basic bioinformatics algorithms.

The second chapter lays the foundations for necessary biological background – central dogma of molecular biology and biological sequences.

Chapters three, four and five focus on important bioinformatics problems and algorithms, implementation and demonstration of each algorithm in the system of several modules written in C++ language. These problems are to find a common motif, longest common subsequence and sequence alignment. Actual running times of implemented algorithms are measured on a computer with two Intel Xeon processors on 2.6 GHz and 8 GB RAM.

Appendix contains detailed reference of implemented system, how to use it and map of files on enclosed CD.

Chapter 1:
Introduction to Bioinformatics

This chapter introduces the term *Bioinformatics* and gives a brief overview of basic biological structures.

1.1 What does *Bioinformatics* stand for?

Bioinformatics is the application of information technology to the field of molecular biology. The term bioinformatics was introduced by Paulien Hogeweg in 1979 for the study of informatics processes in biotic systems. Bioinformatics now entails the creation and advancement of databases, algorithms, computational and statistical techniques, and theories to solve formal and practical problems arising from the management and analysis of biological data.

Over the past few decades rapid developments in genomic and other molecular research technologies and developments in information technologies have combined to produce a tremendous amount of information related to molecular biology. It is the name given to these mathematical and computing approaches used to glean understanding of biological processes. Common activities in bioinformatics include mapping and analyzing DNA and protein sequences, aligning different DNA and protein sequences to compare them and creating and viewing 3-D models of protein structures.

1.2 Biological sequences

The majority of research in bioinformatics focuses on analysis of biological sequences – nucleotide sequences (DNA and RNA) and peptide sequence (proteins).

For a typical unbranched, un-crosslinked biopolymer (such as a molecule of DNA, RNA or protein), the *primary structure* is equivalent to specifying the sequence of its monomeric units, e.g., the nucleotide or peptide sequence. The term "primary structure" was first coined by Linderstrøm-Lang in his 1951 Lane Medical Lectures.

Secondary structure is the general three-dimensional form of local segments of biopolymers such as DNA, RNA or proteins. It does not, however, describe specific atomic positions in three-dimensional space, which are considered to be *tertiary structure*.

1.3 Bioinformatics algorithms

Biological sequences tend to be very long – hence *bioinformatics algorithms* that operate on these sequences must be highly optimized and usually involves brand new approaches and solutions. Nevertheless, sometimes the precision has to be sacrificed to achieve better efficiency.

Vast majority of bioinformatics tools and algorithms performs searching on two or more sequences to find a common motif, longest common subsequence or best alignment.

Motif finding algorithms are usually based on algorithms on trees with branch and bound optimizations; longest common subsequence and sequence alignment algorithms are usually based on dynamic programming approach.

1.4 Secondary structure prediction

New and highly important application of bioinformatics uses predicted RNA secondary structures in searching a genome for non-coding, but functional forms of RNA. A general method of calculating probable RNA secondary structure is dynamic programming, although this has the disadvantage that it can produce biologically ill-formed secondary structures. More general methods are based on stochastic context-free grammars. A web server that implements a type of dynamic programming is Mfold; software package for predicting secondary structures is Vienna RNA Package.

Chapter 2:
Biological Background

This chapter gives a brief overview of basic biological processes, such as transcription and translation.

2.1 Central dogma of molecular biology

The flow of genetic information in the cell is traditionally described in the central dogma of molecular biology.

The genome (DNA) encodes the sequence information for all the proteins synthesized by the cell. The segment of DNA that holds the information of how to construct a certain protein is called a protein-coding gene. However, DNA is not directly the template in the protein synthesis. DNA is transcribed by an enzyme, RNA polymerase, into a messenger RNA (mRNA) carrying the same information as the transcribed gene. The mRNA is then translated into protein by the ribosome. A gene can also be transcribed into an RNA that is never translated into a protein. Such genes are called non-coding genes and the RNAs are called functional or non-coding RNAs (ncRNAs), as they are not translated into protein, but they have a function themselves.

2.1 Biological macromolecules

Deoxyribonucleic acid (DNA) is a nucleic acid that contains the genetic instructions used in the development and functioning of all known living organisms and some viruses. The main role of DNA molecules is the long-term storage of information. DNA is often compared to a set of blueprints or a recipe, or a code, since it contains the instructions needed to construct other components of cells, such as proteins and RNA molecules. The DNA segments that carry this genetic information are called genes, but other DNA sequences have structural purposes, or are involved in regulating the use of this genetic information.

Chemically, DNA consists of two long polymers of simple units called nucleotides (Adenine, Guanine, Cytosine and Thymine), with backbones made of sugars and phosphate groups joined by chemical bonds. These two strands run in opposite directions to each other and are therefore anti-parallel. Attached to each sugar is one of four types of molecules

called bases. It is the sequence of these four bases along the backbone that encodes information. This information is read using the genetic code, which specifies the sequence of the amino acids within proteins. The code is read by copying stretches of DNA into the related nucleic acid RNA, in a process called transcription.

Ribonucleic acid (RNA) – is a biologically important type of molecule that consists of a long chain of nucleotide units. Each nucleotide consists of a nitrogenous base, a ribose sugar, and a phosphate. RNA is very similar to DNA, but differs in a few important structural details: in the cell, RNA is usually single-stranded and RNA has the base Uracil rather than Thymine that is present in DNA.

Proteins (also known as polypeptides) are long polymers made of amino acids arranged in a linear chain. The sequence of amino acids in a protein is defined by the sequence of a gene, which is encoded in the genetic code. In general, the genetic code specifies 20 standard amino acids; however in certain organisms the genetic code can include other amino acids. Proteins can also work together to achieve a particular function, and they often associate to form stable complexes. Very important role of proteins in the cell is as enzymes, which catalyze chemical reactions. Enzymes are usually highly specific and accelerate only one or a few chemical reactions. Enzymes carry out most of the reactions involved in metabolism, as well as manipulating DNA in processes such as DNA replication, DNA repair, and transcription. Some enzymes act on other proteins to add or remove chemical groups in a process known as post-translational modification.

2.2 Transcription

Transcription is the cellular process in which DNA is transcribed by an RNA polymerase to form a complementary RNA copy. The transcription starts at promoters in the DNA. The promoters are certain sequence motifs important for recognition by RNA polymerase. When the RNA polymerase is bound to the promoter, a small part of the DNA helix is unwound and transcription starts. The RNA polymerase reads the DNA sequence, and as it reads the sequence the RNA polymerase synthesizes an RNA molecule complementary to the DNA template. The transcription continues until a terminator sequence is reached.

2.3 Translation

A sequence of three nucleotide bases, a codon, specifies an **amino acid** – the basic structural building units of proteins. There are 22 (20 + 2) proteinogenic (protein building) amino acids, see table 2.1.

Amino Acid	Short	Abbreviation	Amino Acid	Short	Abbreviation
Alanine	A	Ala	Asparagine	N	Asn
Cysteine	C	Cys	Pyrrolysine	O	Pyl
Aspartic acid	D	Asp	Proline	P	Pro
Glutamic acid	E	Glu	Glutamine	Q	Gln
Phenylalanine	F	Phe	Arginine	R	Arg
Glycine	G	Gly	Serine	S	Ser
Histidine	H	His	Threonine	T	Thr
Isoleucine	I	Ile	Selenocysteine	U	Sec
Lysine	K	Lys	Valine	V	Val
Leucine	L	Leu	Tryptophan	W	Trp
Methionine	M	Met	Tyrosine	Y	Tyr

Table 2.1 – proteinogenic amino acids

The genetic code describes the relation (homomorphism) between these tri-nucleotide sequences in the DNA (or mRNA) and the amino acids in the protein sequence, see table 2.2.

UUU Phe	UCU Ser	UAU Tyr	UGU Cys
UUC Phe	UCC Ser	UAC Tyr	UGC Cys
UUA Leu	UCA Ser	UAA stp	UGA stp
UUG Leu	UCG Ser	UAG stp	UGG Trp
CUU Leu	CCU Pro	CAU His	CGU Arg
CUC Leu	CCC Pro	CAC His	CGC Arg
CUA Leu	CCA Pro	CAA Gln	CGA Arg
CUG Leu	CCG Pro	CAG Gln	CGG Arg
AUU Ile	ACU Thr	AAU Asn	AGU Ser
AUC Ile	ACC Thr	AAC Asn	AGC Ser
AUA Ile	ACA Thr	AAA Lys	AGA Arg
AUG Met	ACG Thr	AAG Lys	AGG Arg
GUU Val	GCU Ala	GAU Asp	GGU Gly
GUC Val	GCC Ala	GAC Asp	GGC Gly
GUA Val	GCA Ala	GAA Glu	GGA Gly
GUG Val	GCG Ala	GAG Glu	GGG Gly

Table 2.2 – genetic code [1]

The translation always starts at a specific codon, the start codon AUG, until it reaches one of three stop codons. The translation of the mature mRNA involves several ncRNAs. The translation takes place in the ribosome, a large complex of ribosomal RNAs (rRNAs) and proteins. The codons are read one-by-one by transfer RNAs (tRNAs), adapter molecules carrying specific amino acids that in the ribosome are connected with peptide bonds to form a protein.

2.4 Non–coding RNA

A non-coding RNA (ncRNA) is a functional RNA molecule that is not translated into a protein. For bacterial ncRNAs, the term small RNA (sRNA) is often used. The DNA sequence from which a non-coding RNA is transcribed as the end product is often called an RNA gene or non-coding RNA gene. Example of important ncRNAs is tRNA or rRNA.

The transfer RNA (tRNA) is RNA responsible for bringing the correct amino acid to the ribosome in translation. The tRNA anticodon (inverse of codon) reads off the mRNA codon and brings the amino acid that it is carrying to the ribosome for insertion to a growing peptide (protein).

The ribosomal RNAs (rRNAs) with proteins form the ribosome complex. The rRNAs have a catalytic function (enzyme called ribozymes). The rRNAs are usually the most abundant RNAs in the cell.

Non-coding RNAs are involved in many cellular processes. These range from ncRNAs of central importance that are conserved across all or most cellular life through to more transient ncRNAs specific to one or a few closely related species. The entire branch of bioinformatics studies ncRNAs, mainly the secondary structure and its primary image in the genome.

2.5 Secondary structure of RNA

Nucleic acids have secondary structure, most notably single-stranded RNA molecules (ssRNA). RNA secondary structure is generally divided into helices (contiguous base pairs), and various kinds of loops (unpaired nucleotides surrounded by helices). The stem-loop structure is extremely common and is a building block for larger structural motifs such as cloverleaf structures, which are four-helix junctions such as those found in transfer RNA.

For many RNA molecules, the secondary structure is highly important to the correct function of the RNA — often more so than the actual sequence. This fact aids in the analysis of non-coding RNA sometimes termed "RNA genes". RNA secondary structure can be predicted with some accuracy by computer and many bioinformatics applications use some notion of secondary structure in analysis of RNA. Image 2.1 displays secondary structure of tRNA with GAA anticodon.

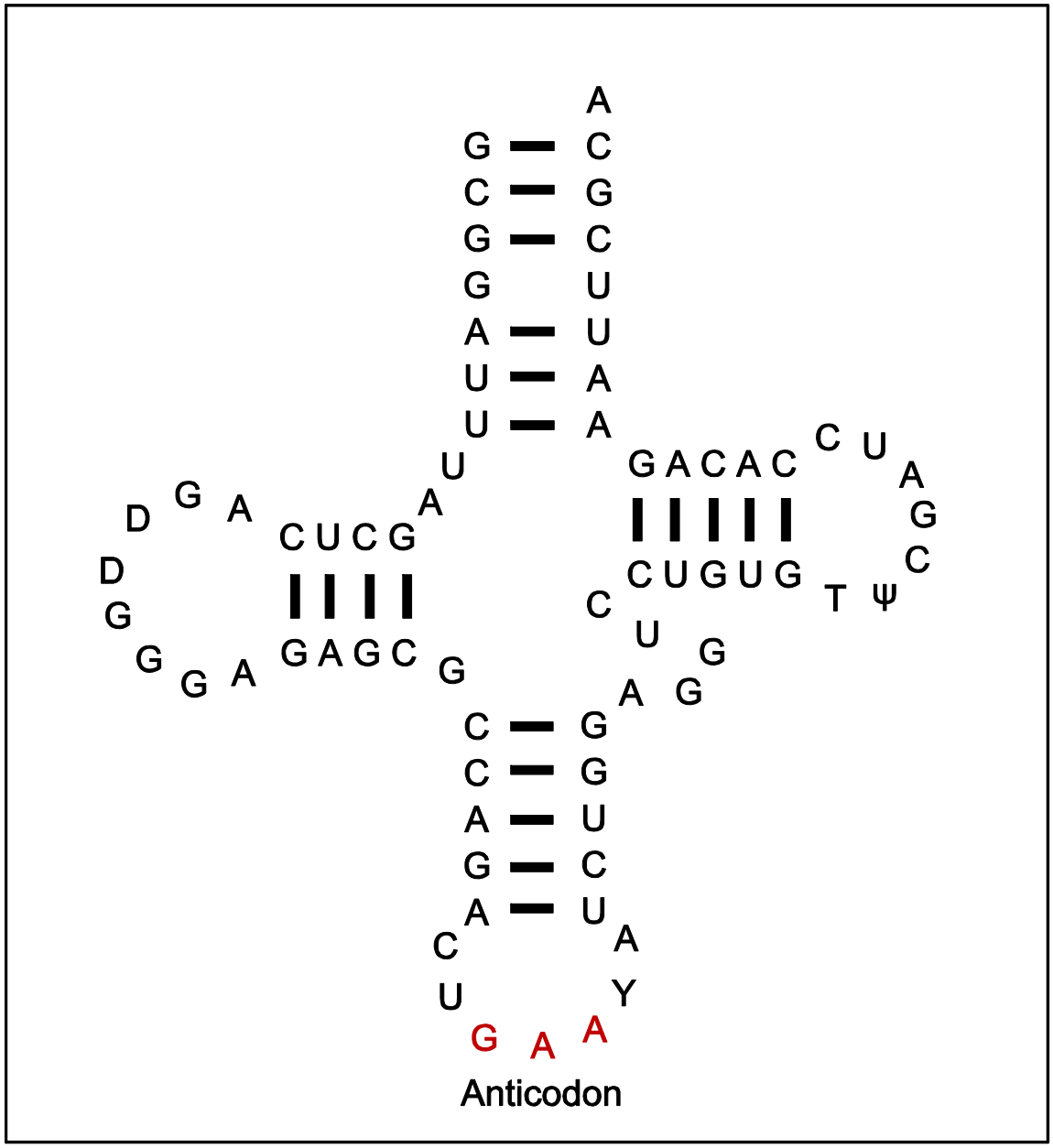


Image 2.1: Secondary structure of tRNA

Chapter 3:
Motif Finding

3.1 Introduction

An important task in unraveling the mechanisms that regulate gene expression is to identify regulatory elements, especially the binding sites in DNA for transcription factors. These binding sites are short DNA segments – they are called motifs.

A DNA motif is defined as a nucleic acid sequence pattern that has some biological significance such as being DNA binding sites for a regulatory protein, i.e., a transcription factor. Normally, the pattern is short (5 to 20 nucleotides long) and is known to recur in different genes or several times within a gene [1].

Motifs have been discovered by studying similar genes in different species. For example, by aligning the amino acid sequences specified by the glial cells (cells in special neural tissue) missing gene in man, mouse and D. melanogaster (small fly), Akiyama [2] and others discovered a pattern which they called the GCM motif. The authors were able to show that the motif has DNA binding activity.

Sequences could have zero, one, or multiple copies of a motif. In addition to the common forms of DNA motifs, two special types of DNA motifs are recognized: palindromic motifs and spaced gapped motifs. A palindromic motif is a subsequence that is exactly the same as its own reverse complement, e.g., CACGTG. A spaced gapped motif consists of two smaller conserved sites separated by a spacer (gap).

Given a set of DNA sequences, the motif finding problem is the task of detecting overrepresented motifs as well as conserved motifs from several orthologous sequences that are good candidates for being transcription factor binding sites.

Large number of motif finding algorithms have been implemented and applied to various motif models over the past decade. This work presents five approaches to solve the motif finding problem.

3.2 Brute force algorithm

The simplest option is to compare each motif of length l from the first sequence with each motif of the same length with all possible motifs in other sequences. If some motif has highest score so far, this score and the positions in all sequences are saved. From practical perspective, this solution is unusable, since the complexity for t sequences with length n is $O(ln^t)$ [7] for motif with length l.

The motif module below implements two types of such brute force algorithm – first algorithm (a1) utilizes recursion, the second algorithm (a2) utilizes tree, constructed from all possible indexes in all sequences – its height is number of sequences and each node has $m - l$ descendents. The path from the root to each leaf represents one permutation of all possible motif indexes. All these motifs are compared and the result is motif with the highest score. This solution again compares all possible motifs against each other, so the complexity is again exponential.

3.3 Median search algorithm

Quite different approach offers median search algorithm. In this case we create all permutations over the alphabet {A, C, T, G} and then we search for highest score for each such permutation on all sequences. The recursive solution offers third implemented algorithm (a3), the solution using tree is implemented in the fourth algorithm (a4). For both cases, the complexity is still exponential [7], but the last solution – median search on the tree – is good for optimization.

3.4 Optimized median search algorithm

Running time of median search algorithm can be optimized by branch-and-bound optimization. Such algorithm is implemented as a fifth algorithm (a5) in the median search module. The key to much better performance is that we don't have to walk through all permutations. When traversing the tree is reached a node with higher hamming distance, whole branch of the tree is skipped and the algorithm continues with the next node on the same level [7].

3.5 The implementation

Motif.exe has a command line interface, when started without parameters, it displays brief user guide. It accepts following parameters and switches:

```
motif -aX length -f file1 file2
motif -aX length -r seqnum seqsize
```

Switch	Meaning
-aX	Specifies what algorithm shall be used -a1: best score, brute force algorithm -a2: best score with tree, brute force algorithm -a3: median string search, brute force algorithm -a4: median string search with tree, brute force algorithm -a5: median string search with tree, optimized algorithm
length	length of the motif to search for
-f	Specifies that two files will be used for input and *file1* and *file2* are paths to these files
-r	Specifies that input data will be randomly generated strings over alphabet {A,C,T,G} and *seqnum* specifies number of sequences, *seqsize* specifies size of each sequence

The module is implemented in file *motif.cpp* and requires to be linked against shared file *utils.cpp*, Visual Studio project file is *motif.vcproj*.

After the launch, all sequences in query are displayed. As a result, the program prints out the positions of the motif in each sequence and again all sequences with highlighted motif (all characters but the motif are lowercase, motif is uppercase). Also progress indication is implemented and when the program is finished, machine time (in milliseconds) is printed.

For testing purposes, several files are available. File fmdv1.txt contains complete genome of "Foot-and-mouth disease virus – type C" [10], file fmdv2.txt contains complete genome of "Foot-and-mouth disease virus C strain C-S8 clone MARLS" [11]. Files *motif1.txt* through *motif5.txt* contains sequences, each has length 68 nucleotides.

To test the first algorithm – brute force algorithm – the program runs with following results:

```
> motif -a1 5 -f motif1.txt motif2.txt motif3.txt motif4.txt

Finding motif of length 5 in 4 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC
AAACGTACGTGCACCCTCTTTCTTCGTGGCTCTGGCCAACGAGGGCTGATGTATAAGACGAAAATTTT
AGCCTCCGATGTAAGTCATAGCTGTAACTATTACCTGCCACCCCTATTACATCTTACGTACGTATACA

Using best score - brute force
```

```
|------------------------------------------------------------|
 ............................................................

Motif is at positions 29, 20, 6, 55:
cctgatagacgctatctggctatccacgTACGTaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgaTACGTacgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaacgTACGTgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatctTACGTacgtataca

Running time: 7894 ms
```

Since the complexity of brute force algorithm is exponential, it's not possible to add more sequences. The exponential complexity of this algorithm is clearly visible on the running time by removing sequences.

Result for three sequences:

```
> motif -a1 5 -f motif1.txt motif2.txt motif3.txt

Finding motif of length 5 in 3 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC
AAACGTACGTGCACCCTCTTTCTTCGTGGCTCTGGCCAACGAGGGCTGATGTATAAGACGAAAATTTT

Using best score - brute force

|------------------------------------------------------------|
 ............................................................

Motif is at positions 26, 21, 3:
cctgatagacgctatctggctatccACGTAcgtaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgatACGTAcgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaACGTAcgtgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt

Running time: 109 ms
```

Result for two sequences:

```
> motif -a1 5 -f motif1.txt motif2.txt

Finding motif of length 5 in 2 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC

Using best score - brute force

|------------------------------------------------------------|
 ............................................................

Motif is at positions 3, 17:
ccTGATAgacgctatctggctatccacgtacgtaggtcctctgtgcgaatctatgcgtttccaaccat
```

```
agtactggtgtacattTGATAcgtacgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc

Running time: 0 ms
```

The second algorithm (switch –a2) gives even worse running times, 156 ms for three sequences and 11 seconds for four sequences:

```
> motif –a2 5 -f motif1.txt motif2.txt motif3.txt motif4.txt

Finding motif of length 5 in 4 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC
AAACGTACGTGCACCCTCTTTCTTCGTGGCTCTGGCCAACGAGGGCTGATGTATAAGACGAAAATTTT
AGCCTCCGATGTAAGTCATAGCTGTAACTATTACCTGCCACCCCTATTACATCTTACGTACGTATACA

Using best score - brute force with tree

Building tree...done
|------------------------------------------------------------|
 ..............................................................

Motif is at positions 26, 21, 3, 56:
cctgatagacgctatctggctatccACGTAcgtaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgatACGTAcgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaACGTAcgtgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatcttACGTAcgtataca

Running time: 11045 ms
```

This worse running time is caused by much higher constant due to more memory requirements when building the tree.

For short motifs, the median string search is far better, because it complexity depends mainly on the length of the motif we're searching for. Unlike algorithms a1 and a2, algorithm a3 can run on five sequences:

```
> motif –a3 5 -f motif1.txt motif2.txt motif3.txt motif4.txt motif5.txt

Finding motif of length 10 in 5 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC
AAACGTACGTGCACCCTCTTTCTTCGTGGCTCTGGCCAACGAGGGCTGATGTATAAGACGAAAATTTT
AGCCTCCGATGTAAGTCATAGCTGTAACTATTACCTGCCACCCCTATTACATCTTACGTACGTATACA
CTGTTATACAACGCGTCATGGCGGGGTATGCGTTTTGGTCGTCGTACGCTCGATCGTTAACGTACGTC

Using median string search - brute force

|----|
 ....

Motif is at positions 26, 21, 3, 56, 42:
```

```
cctgatagacgctatctggctatccACGTACGTAGgtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgatACGTACGTACaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaACGTACGTGCaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatcttACGTACGTATaca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgTCGTACGCTCgatcgttaacgtacgtc

Running time: 23931 ms
```

In this case, length of 10 nucleotides is a practical maximum. Algorithm a4 (tree version of a3) gives very similar results:

```
> motif –a3 5 -f motif1.txt motif2.txt motif3.txt motif4.txt motif5.txt

Finding motif of length 10 in 5 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC
AAACGTACGTGCACCCTCTTTCTTCGTGGCTCTGGCCAACGAGGGCTGATGTATAAGACGAAAATTTT
AGCCTCCGATGTAAGTCATAGCTGTAACTATTACCTGCCACCCCTATTACATCTTACGTACGTATACA
CTGTTATACAACGCGTCATGGCGGGGTATGCGTTTTGGTCGTCGTACGCTCGATCGTTAACGTACGTC

Using median string search - brute force with tree

Building tree...done

|----|
 ....

Motif is at positions 24, 19, 1, 54, 58:
cctgatagacgctatctggctatCCACGTACGTaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgATACGTACGTacaccggcaacctgaaacaaacgctcagaaccagaagtgc
AAACGTACGTgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatcTTACGTACGTataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgtTAACGTACGTc

Running time: 23728 ms
```

The only practically usable algorithm is a5, with branch-and-bound optimization. With same data and same results, the running time is a fraction of the version a4:

```
> motif –a3 5 -f motif1.txt motif2.txt motif3.txt motif4.txt motif5.txt

Finding motif of length 10 in 5 sequences of length 68:
CCTGATAGACGCTATCTGGCTATCCACGTACGTAGGTCCTCTGTGCGAATCTATGCGTTTCCAACCAT
AGTACTGGTGTACATTTGATACGTACGTACACCGGCAACCTGAAACAAACGCTCAGAACCAGAAGTGC
AAACGTACGTGCACCCTCTTTCTTCGTGGCTCTGGCCAACGAGGGCTGATGTATAAGACGAAAATTTT
AGCCTCCGATGTAAGTCATAGCTGTAACTATTACCTGCCACCCCTATTACATCTTACGTACGTATACA
CTGTTATACAACGCGTCATGGCGGGGTATGCGTTTTGGTCGTCGTACGCTCGATCGTTAACGTACGTC

Using median string search - bounded tree

Building tree...done
```

```
|----|
 ....

Motif is at positions 24, 19, 1, 54, 58:
cctgatagacgctatctggctatCCACGTACGTaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgATACGTACGTacaccggcaacctgaaacaaacgctcagaaccagaagtgc
AAACGTACGTgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatcTTACGTACGTataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgtTAACGTACGTc

Running time: 406 ms
```

This algorithm is usable even on very long sequences – on sequences *fmdv1.txt* and *fmdv2.txt* – the length of each sequence is 8115 nucleotides – the running time is around 100 seconds.

Chapter 4:
Longest Common Subsequence

4.1 Introduction

The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (this work focuses only just on two sequences). It is a classic computer science problem and has very important role in bioinformatics.

Mutations are caused either by insertions, deletions, or in-place changes in the DNA strand. The solution of the LCS problem is also a solution for the question "How are these DNA strands similar?"

Formally, we define the common subsequence (which shall not be confused with common substring) of strings $v = v_1 \dots v_n$ and $w = w_1 \dots w_m$ as a two sequences of positions in v and w, $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and $1 \leq j_1 < j_2 < \cdots < j \leq m$, such that the symbols at the corresponding positions in v and w coincide: $v_{i_t} = v_{j_t}$ for $1 \leq t \leq k$ [7]

For example, organism's genome contains sequence "AACCCAAAAAATACGTTTCAG", and then this organism evolves in two branches – in one species, this part of genome undergoes two insertion of "ACG" (e.g., two amino acid is inserted to the protein structure), the other species undergoes deletion of "TTT". The resulting sequences (from the common ancestor) are "AACACGCCAAAACGAAATTTTCAG" and "AACCCAAAAAATACGCAG". Nevertheless, LCS of these two strings tells us that these two DNA strands probably share the same origin.

For arbitrary number of input sequences, the problem is NP-hard [1]. When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming algorithm.

Assuming we have N sequences of lengths n_1, n_2, n_N. A naive approach would test each of the 2^{n_1} subsequences of the first sequence to determine whether they are also subsequences of the remaining sequences; each subsequence may be tested in time linear in the lengths of the remaining sequences, so the time for this algorithm would be $O(2^{n_1} \sum_{i>1} n_i)$. On the other hand, the dynamic programming approach gives a solution in $O(N \prod_{i=1}^{N} n_i)$ – for two sequences of lengths n and m, this gives a complexity of $O(n \times m)$ [4].

There also exist methods with lower complexity [5] – these algorithms often depend on the length of the LCS, the size of the alphabet, or both.

Longest common subsequence is not often unique for given sequences – for example, two sequences "TATGCGTA" and "AACGTGTG" have 4 LCS: {ACGT, AGGT, ATGG, ATGT}. To find all common subsequences of a maximum length, this problem inherently has higher complexity, as the number of such subsequences is exponential in the worst case even for only two input strings [6].

4.2 LCS algorithm

As a first step, the algorithm creates for strings $v = v_1 \dots v_m$ and $w = w_1 \dots w_n$ a matrix $F_{m \times n}$ and individual cells are defined as follows [13]:

For each $1 \leq i \leq m$, $1 \leq j \leq n$, $F_{1,j} = 0, F_{i,1} = 0$ and remaining cells are defined recursively as: if $v_i = w_j$,then $F_{i,j} = F_{i-1,j-1} + 1$, otherwise $F_{i,j} = \max(F_{i,j-1}, F_{i-1,j})$.

The length of longest common subsequence is in the cell $F_{m,n}$. Recursive backtrace begins at $F_{m,n}$ and then continues: if the last characters in the prefixes are equal, they must be in an LCS. If not, check what gave the largest LCS of keeping v_i and w_j, and make the same choice. Just choose one if they were equally long. This way the algorithm returns one selected LCS. To get all LCSs, we have to keep in memory all LCSs found during the backtrace and each time backtrace branches, we'll branch the set of constructed LCSs. In that case, the algorithm is no longer polynomial, since in its worst case it can branch in every cell.

4.3 The implementation

Lcs.exe has a command line interface, when started without parameters, it displays brief user guide. It accepts following parameters and switches:

```
lcs -f file1 file2
lcs -r rand_size1 rand_size2
```

Switch	Meaning
-f	Specifies that two files will be used for input and *file1* and *file2* are paths to these files
-r	Specifies that input data will be randomly generated strings over alphabet {A,C,T,G} and rand_size1 a rand_size2 are sizes of these strings

The module for LCS is implemented in file *lcs.cpp* and requires to be linked against shared file *utils.cpp*, Visual Studio project file is lcs.vcproj.

After the launch, both sequences are displayed. If lengths of both strings are less than 26 characters, scoring matrix is displayed. As a result, the program prints out the length of LCS and number of variations found.

For testing purposes, several files are available. File fmdv1.txt contains complete genome of "Foot-and-mouth disease virus – type C" [10], file fmdv2.txt contains complete genome of "Foot-and-mouth disease virus C strain C-S8 clone MARLS" [11] and file fmdv3.txt contains complete genome of "Foot-and-mouth disease virus O strain Chu-Pei" [12]. File *lcs1.txt* contains short string "`GGGAAGCGGTTCTTTGCGTT`" and file *lcs2.txt* contains short string "`AACAGGGGCGTAAGCCGCTT`".

When running with test files lcs1.txt and lcs2.txt, the LCS and scoring matrix is following:

```
> lcs -f lcs1.txt lcs2.txt

String 1 (20 characters):
GGGAAGCGGTTCTTTGCGTT

String 2 (20 characters):
AACAGGGGCGTAAGCCGCTT

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  0  0  0  0  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
  0  0  0  0  1  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  0  0  0  0  1  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  0  1  1  1  1  2  3  3  4  4  4  4  4  4  4  4  4  4  4  4  4
  0  1  2  2  2  2  3  3  4  5  5  5  5  5  5  5  5  5  5  5  5
  0  1  2  3  3  3  3  3  4  5  5  5  5  5  5  5  6  6  6  6  6
  0  1  2  3  3  3  4  4  4  5  5  5  5  5  5  5  6  6  7  7  7
  0  1  2  3  3  3  4  5  5  5  5  5  6  6  6  6  6  7  7  7  7
  0  1  2  3  3  3  4  5  6  6  6  6  6  6  6  6  7  7  8  8  8
  0  1  2  3  3  3  4  5  6  6  7  7  7  7  7  7  7  7  8  9  9
  0  1  2  3  4  4  4  5  6  6  7  7  7  7  7  7  7  7  8  9  9
  0  1  2  3  4  5  5  5  6  6  7  7  7  7  7  7  7  7  8  9  9
  0  1  2  3  4  5  6  6  6  7  7  7  7  7  7  7  8  8  8  9  9
  0  1  2  3  4  5  6  7  7  7  7  7  8  8  8  8  8  9  9  9  9
  0  1  2  3  4  5  6  7  7  7  7  7  8  8  8  8  8  9  9  9  9
  0  1  2  3  4  5  6  7  8  8  8  8  8  8  8  8  9  9 10 10 10
  0  1  2  3  4  5  6  7  8  8  8  8  9  9  9  9  9 10 10 10 10
  0  1  2  3  4  5  6  7  8  8  9  9  9 10 10 10 10 10 10 11 11
  0  1  2  3  4  5  6  7  8  8  9 10 10 10 11 11 11 11 11 11 12

LCS length: 12, variations: 1
GGGAAGCCGCTT

Matrix creation: 0 ms
LCS generation: 0 ms
```

Using same data, but searching for all LCSs, the result is following:

```
> lcs -f lcs1.txt lcs2.txt -a

String 1 (20 characters):
GGGAAGCGGTTCTTTGCGTT

String 2 (20 characters):
AACAGGGGCGTAAGCCGCTT

  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  0  0  0  0  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
  0  0  0  0  1  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  0  0  0  0  1  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3
  0  1  1  1  1  2  3  3  4  4  4  4  4  4  4  4  4  4  4  4  4
  0  1  2  2  2  2  3  3  4  5  5  5  5  5  5  5  5  5  5  5  5
  0  1  2  3  3  3  3  3  4  5  5  5  5  5  5  5  6  6  6  6  6
  0  1  2  3  3  3  4  4  4  5  5  5  5  5  5  5  6  6  7  7  7
  0  1  2  3  3  3  4  5  5  5  5  5  6  6  6  6  6  7  7  7  7
  0  1  2  3  3  3  4  5  6  6  6  6  6  6  6  6  7  7  8  8  8
  0  1  2  3  3  3  4  5  6  6  7  7  7  7  7  7  7  7  8  9  9
  0  1  2  3  4  4  4  5  6  6  7  7  7  7  7  7  7  7  8  9  9
  0  1  2  3  4  5  5  5  6  6  7  7  7  7  7  7  7  7  8  9  9
  0  1  2  3  4  5  6  6  6  7  7  7  7  7  7  7  8  8  8  9  9
  0  1  2  3  4  5  6  7  7  7  7  7  8  8  8  8  8  9  9  9  9
  0  1  2  3  4  5  6  7  7  7  7  7  8  8  8  8  8  9  9  9  9
  0  1  2  3  4  5  6  7  8  8  8  8  8  8  8  8  9  9 10 10 10
  0  1  2  3  4  5  6  7  8  8  8  8  9  9  9  9  9 10 10 10 10
  0  1  2  3  4  5  6  7  8  8  9  9  9 10 10 10 10 10 10 11 11
  0  1  2  3  4  5  6  7  8  8  9 10 10 10 11 11 11 11 11 11 12

LCS length: 12, variations: 8
AACGGCTGCGTT
AAGGGCTGCGTT
GGGAAGCCGCTT
GGGGCGGCCGTT
GGGGCGGCGCTT
GGGGCGTCCGTT
GGGGCGTCGCTT
GGGGCGTGCGTT

Matrix creation: 0 ms
LCS generation: 78 ms
```

The search for all LCSs took much longer here, since we use recursive, polynomial algorithm.

Interesting results come from LCS of files fmdv1.txt and fmdv2.txt and fmdv1.txt and fmdv3.txt:

```
> lcs -f fmdv1.txt fmdv2.txt

String 1 (8115 characters):
TTGAAAGGGGGCGCTAGGGTCTCACCCCTAACATGCCAACGACAGTCCCTGCATTGCACTCCACACTTACGTTGTGCGTA
CGCGGGGCCCGATGGACCATCGTTCACCCACCTACAGCTGGACTCACGACACCGCGTGGCCATTTTAGCTGGATTGTGCG
GACGAACACCGCTTGCGCTTCTCGCGTGACCGGTTAGTACTCTTACCACCTTCCGCCTACTTGGTTGTTAGCGCTGTCCT
GGGCACCCCTGTTGGGGGCCGTTCGACGCTCCACGGGTTCCCCCGTGCGGCAACTACGGTGATGGGGCCGTTTCGCGCGG
GCTGATCGCCTGGTTTGCTTCGGCTGTCACCCGAAACCCGCCTTTCATAAGTTTTACCGTCTGTCCCGACGTTAAAGGGA
GGTAACCACAAGCATACTGCCGCCTTCCCCGGCGTTAACGGGATGCAACCGTAAGACACACCTTCATCCGGAAGTGAAAC
GGCTGCGTCACATAGTTTTGCCCGTTTTCACGAGAAATGGGACGTCTGCGCACGAAACGCGCCGTCGCTTGAGGAAGACT
TGTACAAACACGATCTAAGCAGGTTTCCCCAACTGACACAAAACGTGCAACTTGAAACTCCGCCTGGTCTTTCCAGGTCT
AGAGGGGTAACACTTTGTACTGTGTTTGGCTCCACGCTCGATCCACTGGCGAGTGTTAGTAACAGCACTGTTGCTTCGTA
GCGGAGCATGACGGCCGTGGGAACTCCTCCTTGGTAACAAGGACCCACGGGGCCAAAAGCCACGCCCACACGGGCCCGTC
ATGTGTGCAACCCCAGCACGGCGACTTTACTGCGAAACCCACTTTAAAGTGACATTGAAACTGGTACCCACACACTGGTG
...
String 2 (8115 characters):
TTGAAAGGGGGCGCTAGGGTCTCACCCCTAAAATGCCAACGACAGTCCCTGCATTGCACTCCATACTTACGTTGTGCGTA
CGCGGGGCCCGATGGACCATCGTTCACCCACCTACAGCTGGACTCACGACACCGCGTGGCCATTTTAGCTGGATTGTGCG
GACGAACACCGCTTGCGCTTCTCGCGTGACCGGTTAGTACTCTTACCACCTTCCGCCTACTTGGTTGTTAGCGCTGTCCT
GGGCACTCCTGTTGGGGGCCGTTCGACGCTCCACGGGTTCCCCCGTGCGGCAACTACGGTGATGGGGCCGTTTCGCGCGG
GCTGATCGCCTGGTTTGCTTCGGCTGTCACCCGAAACCCGCCTTTCATAAGTTTTACCGTCTGTCCCGACGTTAAAGGGA
GGTAACCACAAGCATACTGCCGCCTTCCCCGGCGTTAACGGGATGCAACCGTAAGACACACCTTCATCCGGAAGTAAAAC
GGCTGCGTCACGTAGTTTTGCCCTTTTTTACGAGAAATGGGACGTCTGCGCACGAAACGCGCCGTCGCTTGAGGAAGACT
TGTACAAACACGATCTAAGCAGGTTTCCCCAACTGACACAAAACGTGCAACTTGAAACTCCGCCTGGTCTTTCCAGGTCT
AGAGGGGTAACACTTTGTACTGTGTTTGGCTCCACGCTCGATCCACTGGCGAGTGTTAGTAACAGCACTGTTGCTTCGTA
GCGGAGCATGACGGCCGTGGGAACTCCTCCTTGGTAACAAGGACCCACGGGGCCAAAAGCCACGCCCACACGGGCCCGTC
...
LCS length: 8068, variations: 1
TTGAAAGGGGGCGCTAGGGTCTCACCCCTAAATGCCAACGACAGTCCCTGCATTGCACTCCAACTTACGTTGTGCGTACG
CGGGGCCCGATGGACCATCGTTCACCCACCTACAGCTGGACTCACGACACCGCGTGGCCATTTTAGCTGGATTGTGCGGA
CGAACACCGCTTGCGCTTCTCGCGTGACCGGTTAGTACTCTTACCACCTTCCGCCTACTTGGTTGTTAGCGCTGTCCTGG
GCACCCTGTTGGGGGCCGTTCGACGCTCCACGGGTTCCCCCGTGCGGCAACTACGGTGATGGGGCCGTTTCGCGCGGGCT
GATCGCCTGGTTTGCTTCGGCTGTCACCCGAAACCCGCCTTTCATAAGTTTTACCGTCTGTCCCGACGTTAAAGGGAGGT
AACCACAAGCATACTGCCGCCTTCCCCGGCGTTAACGGGATGCAACCGTAAGACACACCTTCATCCGGAAGTAAACGGCT
GCGTCACTAGTTTTGCCCTTTTACGAGAAATGGGACGTCTGCGCACGAAACGCGCCGTCGCTTGAGGAAGACTTGTACAA
ACACGATCTAAGCAGGTTTCCCCAACTGACACAAAACGTGCAACTTGAAACTCCGCCTGGTCTTTCCAGGTCTAGAGGGG
TAACACTTTGTACTGTGTTTGGCTCCACGCTCGATCCACTGGCGAGTGTTAGTAACAGCACTGTTGCTTCGTAGCGGAGC
ATGACGGCCGTGGGAACTCCTCCTTGGTAACAAGGACCCACGGGGCCAAAAGCCACGCCCACACGGGCCCGTCATGTGTG
...
```

(The rest of the virus RNA code is omitted.)

This LCS shows that these two viruses are nearly equal there are just few mutations between each other.

When we try to find LCS of *fmdv1.txt* and *fmdv3.txt*, the result is following:

```
> lcs -f fmdv1.txt fmdv3.txt

String 1 (8115 characters):
TTGAAAGGGGGCGCTAGGGTCTCACCCCTAACATGCCAACGACAGTCCCTGCATTGCACTCCACACTTACGTTGTGCGTA
CGCGGGGCCCGATGGACCATCGTTCACCCACCTACAGCTGGACTCACGACACCGCGTGGCCATTTTAGCTGGATTGTGCG
GACGAACACCGCTTGCGCTTCTCGCGTGACCGGTTAGTACTCTTACCACCTTCCGCCTACTTGGTTGTTAGCGCTGTCCT
GGGCACCCCTGTTGGGGGCCGTTCGACGCTCCACGGGTTCCCCCGTGCGGCAACTACGGTGATGGGGCCGTTTCGCGCGG
GCTGATCGCCTGGTTTGCTTCGGCTGTCACCCGAAACCCGCCTTTCATAAGTTTTACCGTCTGTCCCGACGTTAAAGGGA
GGTAACCACAAGCATACTGCCGCCTTCCCCGGCGTTAACGGGATGCAACCGTAAGACACACCTTCATCCGGAAGTGAAAC
GGCTGCGTCACATAGTTTTGCCCGTTTTCACGAGAAATGGGACGTCTGCGCACGAAACGCGCCGTCGCTTGAGGAAGACT
TGTACAAACACGATCTAAGCAGGTTTCCCCAACTGACACAAAACGTGCAACTTGAAACTCCGCCTGGTCTTTCCAGGTCT
AGAGGGGTAACACTTTGTACTGTGTTTGGCTCCACGCTCGATCCACTGGCGAGTGTTAGTAACAGCACTGTTGCTTCGTA
GCGGAGCATGACGGCCGTGGGAACTCCTCCTTGGTAACAAGGACCCACGGGGCCAAAAGCCACGCCCACACGGGCCCGTC
ATGTGTGCAACCCCAGCACGGCGACTTTACTGCGAAACCCACTTTAAAGTGACATTGAAACTGGTACCCACACACTGGTG
...
String 2 (7733 characters):
AAGTTTTACCGTCGTTCCCGACGTTTGAAGGGAGGAAACCACACGCTTGCACCACCCCCGGTGTCAACGGGATGCAACCG
CAAGATGGACCTTCGCCCGGAAGTAAAACGGCAGCTTTACACAGTTTTGCCCGTTTTCATGAGAAACGGGACGTCCGCGC
```

```
ACGAAACGCGCCGTCGCTTGAGGAACACTTGTACAAACACGATTTAAGCAGGTTTCCACAACTGATAAAACTCGTGCAAC
TTGAAACCCCGCCTGGTCTTTCCAGGTCTAGAGGGGTGACACTTTGTACTGTGCTCGACTCCACGCCCGGTCCACTGGCG
GGTGTTAGTAGCAGCACTGTTGTTTCGTAGCGGAGCATGGTGGCCGTGGGAACTCCTCCTTGGTGACAAGGGCCCACGGG
GCCGAAAGCCACGTCCAAACGGACCCAACATGTGTGCAACCCCAGCACGGCAACTTTACTGCGAACACCACCTTAAGGTG
ACACTGGTACTGGTACTCGGTCACTGGTGACAGGCTAAGGATGCCCTTCAGGTACCCCGAGGTAACACGGGACACTCGGG
ATCTGAGAAGGGGATTGGGACTTCTTTAAAGTGCCCAGTTTAAAAAGCTTCTACGCCTGAATAGGCGACCGGAGGCCGGC
GCCTTTCCATTACCCACTACTAAATCCATGAATACGACCGACTGCTTTATCGCTCTGCTATACGTTCTCAGAGAGATCAA
AGCACTGTTTCTGTCACGAACACAAGGGAAGATGGAATTCACACTTTACAACGGTGAAAAGAAGGTCTTCCACTCCAGAC
...
LCS length: 6543, variations: 1
AAGTTTTACCGTCGTTCCCACGTTTGAGGGAGGAAACCACACGCTGCACCACCCCCGGTGTCAACGGGTGCACCGCAAGA
TGGACCTTCGCCCGGAAGTAAAACGGCAGCTTTACACAGTTTTGCCCGTTTTCATGAGAAACGGGACGTCCGCGCACGAA
ACGCGCCGTCGCTTGAGGAAACTTGTACAAACACGATTAAGCAGGTTTCCCAACTGAAAAACGTGCAACTTGAAACCCGC
CTGGTCTTTCCAGGTCTAGAGGGGTACACTTTGTACTGTGTGCTCCACGCCGTCCACTGGCGGTGTTAGTACAGCACTGT
TGTTCGTAGCGGAGCATGGGCCGTGGGAACTCCTCCTTGGTACAAGGCCCACGGGGCCAAAGCCACGCCAACGGCCCCAT
GTGTGCAACCCCAGCACGGCACTTTACTGCGAAACCCCTTAAGTGACATTACTGGTACCCACTGGTGACAGGCTAAGGAT
GCCCTTCAGGTACCCCGAGGTAACACGGACACTCGGGATCTGAGAAGGGGATGGGCTTCTTAAAGCCCGTTTAAAAAGCT
TCTAGCCTGAATAGGGACCGGAGGCGGCCCTTTCCTTACAATAATCCATGAATACACGACTGTTTATCGCTTGTAACGCT
CAAAGATAAAGCACTTTTCTCACGACCAGGAAATGGAATTCACCTACAACGGGAAAAAAGGTCTTACTCCAGACCCAACA
ACCACGACAACTGTGGTGAACCCATCCTCATGTTCAGGTACGTCGAGCCTTCTCGATGGGTCTACATCCCGAGAACCTCA
CCTGAGCATCAACAACTGGAGAATCACAGGTTGAGTCCGAGGGGGACCGCCCGCCCTTGTGTTGGAACATCAACACTGCT
...
```

(The rest of the virus RNA code is omitted.)

These two viruses appears to be more different, but LCS with length of 6543 of the total length 7733 (the second virus) still tells us that they share common genome. The actual changes and much more information we get from sequence alignment, especially local alignment.

The commented C++ code of the executive routine (found in source file *lcs.cpp*) follows.

```
void lcs(std::string& string1, std::string& string2, bool findAll)
{
   std::cout << "String 1 (" << string1.length() << " characters):" << std::endl;
   std::cout << string1 << std::endl << std::endl;

   std::cout << "String 2 (" << string2.length() << " characters):" << std::endl;
   std::cout << string2 << std::endl << std::endl;

   int cols = (int)string1.length() + 1;
   int rows = (int)string2.length() + 1;

   // create the matrix
   int **matrix = new int *[rows];
   for (int row = 0; row < rows; row++)
      matrix[row] = new int [cols];

   // init matrix
   for (int row = 0; row < rows; row++)
      matrix[row][0] = 0;

   for (int col = 0; col < cols; col++)
      matrix[0][col] = 0;

   // calculate matrix
   clock_t matrix_start = clock();
   for (int row = 1; row < rows; row++)
   {
      for (int col = 1; col < cols; col++)
      {
         matrix[row][col] = string1[col - 1] == string2[row - 1] ?
                        matrix[row - 1][col - 1] + 1 :
                        max(matrix[row - 1][col], matrix[row][col - 1]);
      }
```

```
    }
    clock_t matrix_end = clock();

    // show matrix (if cols < 26)
    if (cols < 26)
    {
       for (int row = 0; row < rows; row++)
       {
          for (int col = 0; col < cols; col++)
             std::cout << (matrix[row][col] < 10 ? "  " : " ") << matrix[row][col];

          std::cout << std::endl;
       }

       std::cout << std::endl;
    }

    // get paths
    clock_t lcs_start = clock();
    std::set<std::string> paths;

    if (findAll)
    {
       paths = backtrace_all(string1, string2, matrix, rows - 1, cols - 1);
    }
    else
    {
       std::string path;
       backtrace_single(string1, string2, matrix, rows - 1, cols - 1, path);
       paths.insert(path);
    }

    clock_t lcs_end = clock();


    // show paths
    std::cout << "LCS length: " <<  matrix[rows - 1][cols - 1] << ", variations: " <<
       paths.size() << std::endl;

    for (std::set<std::string>::iterator iter = paths.begin();
          iter != paths.end(); iter++)
       std::cout << (*iter) << std::endl;

    std::cout << std::endl;
    std::cout << "Matrix creation: " << (matrix_end - matrix_start) << " ms" << std::endl;
    std::cout << "LCS generation: " << (lcs_end - lcs_start) << " ms" << std::endl;
}
```

Routine above utilizes two functions for backtracking, one for all LCSs, one for single LCS. The routine for all LCSs is based on recursion, so this particular implementation is not suitable for longer sequences (where length > 1000 nucleotides):

```
std::set<std::string> backtrace_all(std::string& string1, std::string& string2, int
**matrix, int rows, int cols)
{
    // Here we're only interested in right bottom corner of the matrix:
    // . . . .
    // . . . .
    // . . a b
    // . . c d
    //
    // if d equals b and/or c, trace back that direction(s)
    // if c equals b, we have a branch
    // if d > c and d > b, it's pivot

    std::set<std::string> paths;
```

```
   if (rows > 0 && cols > 0)
   {
      if (string1[cols - 1] == string2[rows - 1])
      {
         paths = backtrace_all(string1, string2, matrix, rows - 1, cols - 1);

         if (paths.size() == 0)
            paths.insert(std::string());

         for (std::set<std::string>::iterator iter = paths.begin(); iter != paths.end();
                iter++)
            (*iter).push_back(string1[cols - 1]);
      }
      else
      {
         if (matrix[rows][cols - 1] == matrix[rows][cols] && matrix[rows - 1][cols] ==
             matrix[rows][cols])
         {
            // we have a branch
            std::set<std::string> branch1 = backtrace_all(string1, string2, matrix,
               rows - 1, cols);

            paths.insert(branch1.begin(), branch1.end());

            std::set<std::string> branch2 = backtrace_all(string1, string2, matrix,
               rows, cols - 1);

            paths.insert(branch2.begin(), branch2.end());
         }
         else
         {
            if (matrix[rows][cols - 1] == matrix[rows][cols])
            {
               paths = backtrace_all(string1, string2, matrix, rows, cols - 1);
            }
            else if (matrix[rows - 1][cols] == matrix[rows][cols])
            {
               paths = backtrace_all(string1, string2, matrix, rows - 1, cols);
            }
         }
      }
   }

   return paths;
}
```

Chapter 5:
Sequence Alignment

This chapter introduces the concept of Sequence Alignment, its types and summarizes main algorithmic approaches for alignment.

5.1 Introduction

An *alignment* is an alternative representation for differences and similarities between strings. A *sequence alignment* is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. The alignment problem might be also formulated as to find an alignment with minimal Levenshtein distance.

Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. The LCS problem is a specific case of the sequence alignment.

5.2 Global and local alignment

Computational approaches to sequence alignment generally fall into two categories: global alignments and local alignments. Calculating a global alignment is a form of global optimization that "forces" the alignment to span the entire length of all sequences in query. By contrast, local alignments identify regions of similarity within long sequences that are often widely divergent overall. Local alignments are often preferable, but can be more difficult to calculate because of the additional challenge of identifying the regions of similarity. A variety of computational algorithms have been applied to the sequence alignment problem, including slow but formally optimizing methods like dynamic programming, and efficient, but not as thorough heuristic algorithms or probabilistic methods designed for large-scale database search.

Global alignments, which attempt to align every residue in every sequence, are most useful when the sequences in the query set are similar and of roughly equal size. A general global alignment technique is the Needleman-Wunsch algorithm, which is based on dynamic programming. Local alignments are more useful for dissimilar sequences that are suspected

to contain regions of similarity or similar sequence motifs within their larger sequence context. The Smith-Waterman algorithm is a general local alignment method also based on dynamic programming. With sufficiently similar sequences, there is no difference between local and global alignments.

5.3 Multiple sequence alignment

A multiple sequence alignment is a sequence alignment of three or more biological sequences. In many cases, the input set of query sequences are assumed to have an evolutionary relationship by which they share a lineage and are descended from a common ancestor. From the resulting alignment, sequence homology can be inferred and phylogenetic analysis can be conducted to assess the sequences' shared evolutionary origins. Visual depictions of the alignment as in the image at right illustrate mutation events such as point mutations (single amino acid or nucleotide changes) that appear as differing characters in a single alignment column, and insertion or deletion mutations that appear as hyphens in one or more of the sequences in the alignment. Multiple sequence alignment is often used to assess sequence conservation of protein domains, tertiary and secondary structures, and even individual amino acids or nucleotides.

5.4 Brute force approach

A brute force approach for finding the best alignment is to try all possible alignments and keep the one with the best score. This can be achieved by trying all three possibilities at each position – no gap, insert a gap for the first sequence, insert a gap for the second sequence. Brute force method has exponential complexity (as explained in chapter 4 with LCS), therefore it's too computationally expensive for anything but short sequences.

5.5 Dynamic programming approach

The most common dynamic programming algorithm for global alignment is Needleman–Wunsch algorithm, which performs a global alignment on two sequences. The algorithm was published in 1970 by Saul Needleman and Christian Wunsch [8].

Unlike in LCS, the similarity matrix contains both positive and negative numbers; the algorithm also works with a linear gap penalty (usually negative number). The example of similarity matrix is in table 5.1. This table tells the algorithm that match A–A is highly favorite and C–G is the least likely to occur.

-	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

Table 5.1: example of similarity matrix

In practice, there are many different similarity matrices for specific applications.

As a first step, the algorithm creates for strings $v = v_1 \dots v_m$ and $w = w_1 \dots w_n$ a matrix $F_{m \times n}$ and individual cells are defined as follows:

For each $1 \leq i \leq m$, $1 \leq j \leq n$, $F_{1,j} = gap * j, F_{i,1} = gap * i$ and remaining cells are defined recursively as $F_{i,j} = \max(F_{i-1,j-1} + S_{v_i,w_j}, F_{i,j-1} + gap, F_{i-1,j} + gap)$ where gap is a linear gap penalty and S is a scoring function for string's alphabet.

Traceback begins at $F_{m,n}$ where both sequences are globally aligned. At each cell, we look to see where we move next according to the pointers.

To find local alignment, this algorithm needs to be modified, according to Smith-Waterman [9]: When a value in the score matrix becomes negative, reset it to zero (begin of new alignment), so the maximizing function from previous definition becomes modified as follows: $F_{i,j} = \max(F_{i-1,j-1} + S_{v_i,w_j}, F_{i,j-1} + gap, F_{i-1,j} + gap, 0)$. The trackback begins in the cell (or cells) with the highest score.

The running time for both algorithms is $O(m \times \mathrm{n})$ [8], [9].

5.6 The implementation

Align.exe has a command line interface, when started without parameters, it displays brief user guide. It accepts following parameters and switches:

```
align -f [-l] gap file1 file2
align -r [-l] gap rand_size1 rand_size2
```

Switch	Meaning
-f	Specifies that two files will be used for input and file1 and file2 are paths to these files
-r	Specifies that input data will be randomly generated strings over alphabet {A,C,T,G} and rand_size1 a rand_size2 are sizes of these strings
-l	Specifies that local alignment shall be performed instead of global alignment
gap	Whole number of gap penalty, usually negative, recommended value is –5

The module for alignment is implemented in file *align.cpp* and requires to be linked against shared file *utils.cpp*, Visual Studio project file is align.vcproj.

After the launch, both sequences are displayed. If lengths of both strings are less than 18 characters, scoring matrix is displayed. As a result, the program prints out the highest score and both aligned strings. Matching characters are uppercase; mismatches are denoted by lowercase letters, gaps by dashes. For example, the alignment of strings "AGTAAACTGCCGTAC" and "AGTACTGGACTGCAT" would appear as:

```
AGTaaACT-GcC-GtAc
AGT--ACTgGaCtGcAt
```

If both sequences are longer than 70 characters, the result is displayed by lines, where first line comes from the first string and the second line comes from the second string, characters are grouped into 10-tuples – therefore, the resulting alignment is clearly visible:

```
aGaCaG-AAGA tGGTggA-tg ATgCGGTGaA TGAgtaCAtC gaGaaaGcaa ACatCaccAc CGaTgaccA
cG-C-GcAAGA -GGT--Acca AT-CGGTGgA TGA-ccCA-C t-G---G--- ACggCg--A- CG-T----A

gaGaaaGcaaA CatCaccAcC GaTgaccAGa CaCTTGaCGA gGCGGAAAAG AACCCTCTGG AaACcAGTG
t-G---G---A CggCg--A-C G-T----AG- CtCTTGgCGA cGCGGAAAAG AACCCTCTGG AgACgAGTG
```

For testing purposes, several files are available. File fmdv1.txt contains complete genome of "Foot-and-mouth disease virus – type C" [10], file fmdv2.txt contains complete genome of "Foot-and-mouth disease virus C strain C-S8 clone MARLS" [11] and file fmdv3.txt contains complete genome of "Foot-and-mouth disease virus O strain Chu-Pei" [12]. File *align1.txt* contains short string "AGTAAACTGCCGTAC" and file *align2.txt* contains short string "AGTACTGGACTGCAT".

When running with test files align1.txt and align2.txt, the global alignment is following:

```
> align -f -5 align1.txt align2.txt

String 1:
AGTAAACTGCCGTAC

String 2:
AGTACTGGACTGCAT

      e   A   G   T   A   A   A   C   T   G   C   C   G   T   A   C
    ---------------------------------------------------------------
e |   0  -5 -10 -15 -20 -25 -30 -35 -40 -45 -50 -55 -60 -65 -70 -75
A |  -5  10   5   0  -5 -10 -15 -20 -25 -30 -35 -40 -45 -50 -55 -60
G | -10   5  19  14   9   4  -1  -6 -11 -16 -21 -26 -31 -36 -41 -46
T | -15   0  14  27  22  17  12   7   2  -3  -8 -13 -18 -23 -28 -33
A | -20  -5   9  22  37  32  27  22  17  12   7   2  -3  -8 -13 -18
C | -25 -10   4  17  32  36  31  34  29  24  19  14   9   4  -1  -6
T | -30 -15  -1  12  27  31  32  29  42  37  32  27  22  17  12   7
G | -35 -20  -6   7  22  26  28  27  37  51  46  41  36  31  26  21
G | -40 -25 -11   2  17  21  23  23  32  46  46  41  50  45  40  35
A | -45 -30 -16  -3  12  27  31  26  27  41  45  45  45  46  55  50
C | -50 -35 -21  -8   7  22  26  38  33  36  48  52  47  42  50  62
T | -55 -40 -26 -13   2  17  21  33  46  41  43  47  52  55  50  57
G | -60 -45 -31 -18  -3  12  16  28  41  55  50  45  56  52  52  52
C | -65 -50 -36 -23  -8   7  11  23  36  50  62  57  52  53  51  59
A | -70 -55 -41 -28 -13   2  17  18  31  45  57  61  56  51  63  58
T | -75 -60 -46 -33 -18  -3  12  14  26  40  52  56  61  64  59  60

Highest score:
60

Alignment:
AGTaaACT-GcC-GtAc
AGT--ACTgGaCtGcAt
```

Using same data, but searching for local alignment, the result is different:

```
> align -f –l -5 align1.txt align2.txt

String 1:
AGTAAACTGCCGTAC

String 2:
AGTACTGGACTGCAT

      e   A   G   T   A   A   A   C   T   G   C   C   G   T   A   C
    ---------------------------------------------------------------
e |   0  -5 -10 -15 -20 -25 -30 -35 -40 -45 -50 -55 -60 -65 -70 -75
A |  -5  10   5   0   0   0   0   0   0   0   0   0   0   0   0   0
G | -10   5  19  14   9   4   0   0   0   9   4   0   9   4   0   0
T | -15   0  14  27  22  17  12   7   8   4   6   1   4  17  12   7
A | -20   0   9  22  37  32  27  22  17  12   7   5   0  12  27  22
C | -25   0   4  17  32  36  31  34  29  24  19  14   9   7  22  34
T | -30   0   0  12  27  31  32  29  42  37  32  27  22  17  17  29
```

```
G |  -35   0   9   7  22  26  28  27  37  51  46  41  36  31  26  24
G |  -40   0   9   9  17  21  23  23  32  46  46  41  50  45  40  35
A |  -45   0   4   5  19  27  31  26  27  41  45  45  45  46  55  50
C |  -50   0   0   1  14  22  26  38  33  36  48  52  47  42  50  62
T |  -55   0   0   8   9  17  21  33  46  41  43  47  52  55  50  57
G |  -60   0   9   4   5  12  16  28  41  55  50  45  56  52  52  52
C |  -65   0   4   6   3   7  11  23  36  50  62  57  52  53  51  59
A |  -70   0   0   1  16  13  17  18  31  45  57  61  56  51  63  58
T |  -75   0   0   8  11  12  12  14  26  40  52  56  61  64  59  60

Highest score:
60

Alignment:
AGTAa---ACTGCcgTAC
AGTActggACTGCa-T--
```

Under normal conditions, these algorithms align much longer sequences; with these two strings it was possible to print out the resulting matrices. With benefit of dynamic programming approach, it takes just few moments for even much longer sequences to align. Interesting results come from alignment of files fmdv1.txt and fmdv3.txt:

```
Highest score:
48968

Alignment:
TTGAAAggggg cgctAggGTc TcaccccTaa caTgccaACg aCaGTccCtG catTgcacTc CaCaCttAC
-----A----- ----A--GT- T------T-- --T----AC- -C-GT--C-G ---T----T- C-C-C-gAC

aCaGTccCtGc atTgcacTcC aCaCttACGT TgTgcGtAcg cggGGcccGA tGGAccAtcg ttcAcCCAc
-C-GT--C-G- --T----T-C -C-C-gACGT T-T--G-A-- a--GG---GA -GGA--A--- ---A-CCA-

cggGGcccGAt GGAccAtcgt tcAcCCAcCt ACaGCTgGac tCACgaCAcC gCgtggCCat tttagCtGG
a--GG---GA- GGA--A---- --A-CCA-C- AC-GCTtG-- -CAC--CA-C -C----CC-- -----C-GG

tCACgaCAcCg CgtggCCatt ttagCtGGat TGTgcggaCg aAcAccgctt gcgcttctCG cGtGAccgg
-CAC--CA-C- C----CC--- ----C-GG-- TGT-----C- -A-A------ --------CG -G-GA----

aAcAccgcttg cgcttctCGc GtGAccggtT aGtactCttA ccAccttCCG cCtActtggt tgttAgcgc
-A-A------- -------CG- G-GA-----T -G----C--A --A----CCG -C-A------ ----A----

ccAccttCCGc CtActtggtt gttAgcgctg tcctggGcAc ccctgtTggg ggccGttcGA cgctcCaCg
--A----CCG- C-A------- ---A------ ------G-A- ------T--- ----G---GA -----C-C-

ccctgtTgggg gccGttcGAc gctcCaCggg tTcccccgTg CgGCaaCtaC gGtGAtgggg ccgtttcgc
------T---- ---G---GA- ----C-C--- -T------T- C-GC--C--C -G-GA----- ---------

CgGCaaCtaCg GtGAtggggc cgtttcgcgc gggctgAtcg cctggtttgc ttcggctGTc AcccgAAAc
C-GC--C--C- G-GA------ ---------- ------A--- ---------- -------GT- A----AAA-

cctggtttgct tcggctGTcA cccgAAAccC GcctttCata agttttAccg tctgtcccGa CgTTaaagg
---------- ------GT-A ----AAA--C G----gC--- ------A--- --------G- C-TT-----

agttttAccgt ctgtcccGaC gTTaaaggga ggTaaccACa AgCatActgc cGcctTcccc ggcgTTaac
------A---- -------G-C -TT------- --T----AC- A-C--A---- -G---T---- ----TT---
```

```
AgCatActgcc GcctTccccg gcgTTaacgg gaTGCaaCCG Taagacacac cTTCATccgG aAgtGaAAc
A-C--A----- G---T----- ---TT----- --TGC--CCG T--t------ -TTCAT---G -A--G-AA-

Taagacacacc TTCATccgGa AgtGaAAcgg ctgcgtcACa taGttttGcc cGttttcACg aGaaaTggg
T--t------- TTCAT---G- A--G-AA--- -------AC- --G----G-- -G-----AC- -G---T---

taGttttGccc GttttcACga GaaaTgggaC gtCtGCGCAC GAAACGCGCC GTCGCTTGAG GAAgACTTG
--G----G--- G-----AC-- G---T----C --C-GCGCAC GAAACGCGCC GTCGCTTGAG GAAcACTTG

GAAACGCGCCG TCGCTTGAGG AAgACTTGTA CAAACACGAT cTAAGCAGGT TTCCcCAACT GAcAcAAAa
GAAACGCGCCG TCGCTTGAGG AAcACTTGTA CAAACACGAT tTAAGCAGGT TTCCaCAACT GAtA-AAAc

cTAAGCAGGTT TCCcCAACTG AcAcAAAa-C GTGCAACTTG AAACtCCGCC TGGTCTTTCC AGGTCTAGA
tTAAGCAGGTT TCCaCAACTG AtA-AAActC GTGCAACTTG AAACcCCGCC TGGTCTTTCC AGGTCTAGA

AAACtCCGCCT GGTCTTTCCA GGTCTAGAGG GGTaACACTT TGTACTGTGt TtGgCTCCAC GCtCGaTCC
AAACcCCGCCT GGTCTTTCCA GGTCTAGAGG GGTgACACTT TGTACTGTGc TcGaCTCCAC GCcCGgTCC

TGTACTGTGtT tGgCTCCACG CtCGaTCCAC TGGCGaGTGT TAGTAaCAGC ACTGTTGcTT CGTAGCGGA
TGTACTGTGcT cGaCTCCACG CcCGgTCCAC TGGCGgGTGT TAGTAgCAGC ACTGTTGtTT CGTAGCGGA

TAGTAaCAGCA CTGTTGcTTC GTAGCGGAGC ATGacGGCCG TGGGAACTCC TCCTTGGTaA CAAGGaCCC
TAGTAgCAGCA CTGTTGtTTC GTAGCGGAGC ATGgtGGCCG TGGGAACTCC TCCTTGGTgA CAAGGgCCC

TGGGAACTCCT CCTTGGTaAC AAGGaCCCAC GGGGCCaAAA GCCACGcCCA cACGGgCCCg tCATGTGTG
TGGGAACTCCT CCTTGGTgAC AAGGgCCCAC GGGGCCgAAA GCCACGtCCA aACGGaCCCa aCATGTGTG
```

(The rest of the virus RNA code is omitted.)

This alignment shows that the beginning of this virus has undergone severe mutation, whereas the rest of the virus remains almost unchanged.

When running these two files with local alignment option, the result instantly shows how the virus has changed:

```
Highest score:
50562

Alignment:
TTGAAAGGGGG CGCTAGGGTC TCACCCCTAA CATGCCAACG ACAGTCCCTG CATTGCACTC CACACTTAC
----------- ---------- ---------- ---------- ---------- ---------- ---------

ACAGTCCCTGC ATTGCACTCC ACACTTACGT TGTGCGTACG CGGGGCCCGA TGGACCATCG TTCACCCAC
----------- ---------- ---------- ---------- ---------- ---------- ---------

CGGGGCCCGAT GGACCATCGT TCACCCACCT ACAGCTGGAC TCACGACACC GCGTGGCCAT TTTAGCTGG
----------- ---------- ---------- ---------- ---------- ---------- ---------

TCACGACACCG CGTGGCCATT TTAGCTGGAT TGTGCGGACG AACACCGCTT GCGCTTCTCG CGTGACCGG
----------- ---------- ---------- ---------- ---------- ---------- ---------

AACACCGCTTG CGCTTCTCGC GTGACCGGTT AGTACTCTTA CCACCTTCCG CCTACTTGGT TGTTAGCGC
----------- ---------- ---------- ---------- ---------- ---------- ---------

CCACCTTCCGC CTACTTGGTT GTTAGCGCTG TCCTGGGCAC CCCTGTTGGG GGCCGTTCGA CGCTCCACG
----------- ---------- ---------- ---------- ---------- ---------- ---------

CCCTGTTGGGG GCCGTTCGAC GCTCCACGGG TTCCCCCGTG CGGCAACTAC GGTGATGGGG CCGTTTCGC
----------- ---------- ---------- ---------- ---------- ---------- ---------
```

```
CGGCAACTACG GTGATGGGGC CGTTTCGCGC GGGCTGATCG CCTGGTTTGC TTCGGCTGTC ACCCGAAAC
----------- ---------- ---------- ---------- ---------- ---------- ---------

CCTGGTTTGCT TCGGCTGTCA CCCGAAACCC GCCTTTCATA -AGTTTTACC GTCtgTCCCG ACG-TTaAA
----------- ---------- ---------- ---------- aAGTTTTACC GTCgtTCCCG ACGtTTgAA

-AGTTTTACCG TCtgTCCCGA CG-TTaAAGG GAGGtAACCA CAaGCaTacT GCcgCCttCC CCGGcGTtA
aAGTTTTACCG TCgtTCCCGA CGtTTgAAGG GAGGaAACCA CAcGC-T--T GCa-CCacCC CCGGtGTcA

CAaGCaTacTG CcgCCttCCC CGGcGTtAAC GGGATGCAAC CGtAAGAcac ACCTTCatCC GGAAGTgAA
CAcGC-T--TG Ca-CCacCCC CGGtGTcAAC GGGATGCAAC CGcAAGAtgg ACCTTCgcCC GGAAGTaAA

CGtAAGAcacA CCTTCatCCG GAAGTgAAAC GGCtGCgTcA CAtAGTTTTG CCCGTTTTCA cGAGAAAtG
CGcAAGAtggA CCTTCgcCCG GAAGTaAAAC GGCaGCtTtA CAcAGTTTTG CCCGTTTTCA tGAGAAAcG

CAtAGTTTTGC CCGTTTTCAc GAGAAAtGGG ACGTCtGCGC ACGAAACGCG CCGTCGCTTG AGGAAgACT
CAcAGTTTTGC CCGTTTTCAt GAGAAAcGGG ACGTCcGCGC ACGAAACGCG CCGTCGCTTG AGGAAcACT

ACGAAACGCGC CGTCGCTTGA GGAAgACTTG TACAAACACG ATcTAAGCAG GTTTCCcCAA CTGAcAcAA
ACGAAACGCGC CGTCGCTTGA GGAAcACTTG TACAAACACG ATtTAAGCAG GTTTCCaCAA CTGAtA-AA

ATcTAAGCAGG TTTCCcCAAC TGAcAcAAAa -CGTGCAACT TGAAACtCCG CCTGGTCTTT CCAGGTCTA
ATtTAAGCAGG TTTCCaCAAC TGAtA-AAAc tCGTGCAACT TGAAACcCCG CCTGGTCTTT CCAGGTCTA
```

(The rest of the virus RNA code is omitted.)

It's obvious how the virus changed – first ~600 nucleotides of the second virus were deleted.

The commented C++ code of the executive routine (found in source file *align.cpp*) follows.

```
void align(std::string& string1, std::string& string2)
{
   std::cout << "String 1:" << std::endl;
   std::cout << string1 << std::endl << std::endl;

   std::cout << "String 2:" << std::endl;
   std::cout << string2 << std::endl << std::endl;

   int cols = (int)string1.length() + 1;
   int rows = (int)string2.length() + 1;

   // create the matrix
   int **matrix = new int *[rows];
   for (int row = 0; row < rows; row++)
      matrix[row] = new int [cols];

   // init matrix
   for (int row = 0; row < rows; row++)
      matrix[row][0] = row * gap;

   for (int col = 0; col < cols; col++)
      matrix[0][col] = col * gap;

   // calculate matrix
   clock_t matrix_start = clock();
   for (int row = 1; row < rows; row++)
   {
      for (int col = 1; col < cols; col++)
      {
         matrix[row][col] = max3
         (
                              matrix[row - 1][col - 1] +
```

```
                           simatrix[c2n(string1[col - 1])][c2n(string2[row - 1])],
            matrix[row - 1][col] + gap,
            matrix[row][col - 1] + gap
         );

         if (local)
         {
            matrix[row][col] = max(0, matrix[row][col]);
         }
      }
   }
   clock_t matrix_end = clock();

   // show matrix (if cols < 19)
   if (cols < 19)
   {
      for (int row = 0; row < rows; row++)
      {
         if (row == 0)
         {
            std::cout << "    ";

            for (int col = 0; col < cols; col++)
               std::cout << "   " <<
                                    (col > 0 ? string1[col - 1] : 'e');

            std::cout << std::endl;

            std::cout << "    ";

            for (int col = 0; col < cols; col++)
               std::cout << "----";

            std::cout << std::endl;
         }

         for (int col = 0; col < cols; col++)
         {
            if (col == 0)
            {
               std::cout << (row > 0 ? string2[row - 1] : 'e');
               std::cout << " | ";
            }

            if (matrix[row][col] >= 10)
               std::cout << "  ";
            else if (matrix[row][col] >= 0)
               std::cout << "   ";
            else if (matrix[row][col] > -10)
               std::cout << "  ";
            else
               std::cout << " ";

            std::cout << matrix[row][col];
         }

         std::cout << std::endl;
      }

      std::cout << std::endl;
   }

   // print score
   std::cout << "Highest score: " << std::endl;
   std::cout << matrix[rows - 1][cols - 1] << std::endl;
   std::cout << std::endl;

   // print alignment
   std::string a;
   std::string b;
```

```
int r = rows - 1;
int c = cols - 1;

// if local alignment is used, find the highest cell
if (local)
{
   int highest_score = 0;
   int highest_r = r;
   int highest_c = c;

   for (int row = 0; row < rows; row++)
   {
      for (int col = 0; col < cols; col++)
      {
         if (highest_score < matrix[row][col])
         {
            highest_score = matrix[row][col];
            highest_r = row;
            highest_c = col;
         }
      }
   }

   while (c > highest_c)
   {
      a.insert(a.begin(), string1[c - 1]);
      b.insert(b.begin(), '-');
      c--;
   }

   while (r > highest_r)
   {
      a.insert(a.begin(), '-');
      b.insert(b.begin(), string2[r - 1]);
      r--;
   }
}

while (r > 0 && c > 0)
{
   int score = matrix[r][c];
   int diagonal = matrix[r - 1][c - 1];
   int up = matrix[r - 1][c];
   int left = matrix[r][c - 1];

   if (score == diagonal + simatrix[c2n(string1[c - 1])][c2n(string2[r-1])])
   {
      bool equal = string1[c - 1] == string2[r - 1];
      a.insert(a.begin(), equal ? string1[c-1] : tolower(string1[c-1]));
      b.insert(b.begin(), equal ? string2[r-1] : tolower(string2[r-1]));
      c--;
      r--;
   }
   else if (score == left + gap)
   {
      a.insert(a.begin(), tolower(string1[c - 1]));
      b.insert(b.begin(), '-');
      c--;
   }
   else // (score == up + gap)
   {
      a.insert(a.begin(), '-');
      b.insert(b.begin(), tolower(string2[r - 1]));
      r--;
   }

   if (local && score == 0)
      break;
}

while (c > 0)
```

```
    {
       a.insert(a.begin(), string1[c - 1]);
       b.insert(b.begin(), '-');
       c--;
    }

    while (r > 0)
    {
       a.insert(a.begin(), '-');
       b.insert(b.begin(), string2[r - 1]);
       r--;
    }

    std::cout << "Alignment:"  << std::endl;

    if (a.length() > 80)
    {
       for (unsigned i = 0; i < a.length(); i++)
       {
          for (unsigned j = 0; j < 70 && (i + j < a.length()); j++)
          {
             std::cout << a[i + j];
             if (j > 0 && j % 10 == 0)
                std::cout << " ";
          }

          std::cout << std::endl;

          for (unsigned j = 0; j < 70 && (i + j < b.length()); j++)
          {
             std::cout << b[i + j];
             if (j > 0 && j % 10 == 0)
                std::cout << " ";
          }

          std::cout << std::endl;
          std::cout << std::endl;

          i += 40;
       }
    }
    else
    {
       std::cout << a << std::endl;
       std::cout << b << std::endl;
    }
}
```

Conclusion

The goal of this work was to explore and study basic methods and algorithms of bioinformatics.

Chapter 2 laid the basic biological grounds that were necessary for further chapters – central dogma of molecular biology (DNA → mRNA → tRNA + ribosome → protein), important biological macromolecules (DNA, RNA and proteins), mechanism of translation, transcription, ncRNAs, and higher structures of RNA.

Chapter 3 presented Motif finding problem: to find the most similar motif with given length on given sequences – from naïve brute force algorithm with exponential complexity to optimized branch-and-bound solution.

Chapter 4 explained and demonstrated the longest common subsequence problem: LCS is a focal point of many important bioinformatics problems and applications; it's also very important example of dynamic programming approach to the algorithm design.

Chapter 5 studied sequence alignment in close detail, discussed scoring matrices and showed the difference between local and global alignment and the use of these algorithms in the field research.

The integral part of this work is a system of four modules – *motif.cpp*, *lcs.cpp*, *align.cpp* and *utils.cpp*. These modules implement eight important bioinformatics algorithms – five for motif finding problem (*motif.cpp)*, the LCS (*lcs.cpp*) problem and algorithms for global and local alignment (*align.cpp*). Module *utils.cpp* contains useful functions and is utilized by other delivered modules. User documentation is included in each chapter that introduces specific module; development documentation is included both in the source code and in the appendix.

Bibliography

[1] Rombauts S, Dehais P, Van Montagu M, Rouze P: *PlantCARE, a plant cis acting regulatory element database*, Nucleic Acids Res, 1999

[2] Akiyama, Hosoya, Poole, Hotta: *The gcm-motif: a novel DNA-binding motif conserved in Drosophila and mammals*, Proc Natl Acad Sci USA, 1996

[3] Ridley, M.: *Genome,* Harper Perennial, 2006

[4] David Maier: *The Complexity of Some Problems on Subsequences and Supersequences,* ACM Press, 1978

[5] L. Bergroth and H. Hakonen and T. Raita: *A Survey of Longest Common Subsequence Algorithms,* IEEE Computer Society, 2000

[6] Ronald I. Greenberg: *Bounds on the Number of Longest Common Subsequences*, 2003

[7] Neil C. Jones, Pavel A. Pevzner: *The Introduction to Bioinformatics Algorithms*, MIT Press, 2004

[8] Needleman, Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, J Mol Biol , 1970

[9] Smith, Waterman: *Identification of Common Molecular Subsequences*, Journal of Molecular Biology, 1981

[10] Herrera M., EMBL/GenBank/DDBJ databases, 2006

[11] Baranowski E., Sevilla N., Verdaguer N., Ruiz-Jarabo C.M., Beck E., Domingo E.: Multiple virulence determinants of foot-and-mouth disease virus in cell culture, J. Virol., 1998

[12] Tsai C., Pan C., Liu M., Lin Y., Chen C., Huang T., Cheng I., Jong M.,Yang P.: Molecular epidemiological studies on foot-and-mouth disease type O Taiwan viruses from the 1997 epidemic, Vet. Microbiol., 2000

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*, Second Edition, MIT Press, 2001

Appendix

This chapter serves as a reference library for presented software package.

A.1 Contents of enclosed CD-ROM

The enclosed CD-ROM contains following files:

- align.cpp: C++ source code for alignment module
- align.exe: executable file for alignment (consists of align.cpp and utils.cpp)
- align.vcproj: Visual Studio 2005 project file for alignment module
- align1.txt: test data file for alignment
- align2.txt: test data file for alignment
- bioinf.sln: Visual Studio 2005 solution for the entire bioinformatics system
- fmdv1.txt: test data file for alignment and LCS (FMDV virus)
- fmdv2.txt: test data file for alignment and LCS (FMDV virus)
- fmdv3.txt: test data file for alignment and LCS (FMDV virus)
- lcs.cpp: C++ source code for LCS module
- lcs.exe: executable file for LCS (consists of lcs.cpp and utils.cpp)
- lcs.vcproj: Visual Studio 2005 project file for LCS module
- lcs1.txt: test data file for LCS
- lcs2.txt: test data file for LCS
- motif.cpp: C++ source code for motif finding module
- motif.exe: executable file for motif finding (consists of motif.cpp and utils.cpp)
- motif.vcproj: Visual Studio 2005 project file for motif finding module
- motif1.txt: test data file for motif finding
- motif2.txt: test data file for motif finding
- motif3.txt: test data file for motif finding
- motif4.txt: test data file for motif finding
- motif5.txt: test data file for motif finding
- Thesis.pdf: this thesis in PDF
- utils.cpp: C++ source code for function shared by all modules
- utils.h: header file for utils.cpp

A.2 Reference of module motif.cpp

```
struct Node
{
   Node(Node *parent, int depth, int length, int value);
   ~Node();

   Node *FirstLeaf();
   Node *NextLeaf();
   Node *NextVertex(bool fromBottom = false);
   Node *BypassVertext();
   Node *Bypass();

   // assuming length == 4, returns path in ACGT
   void String(std::string& string);
   void Values(std::vector<unsigned>& values);
};
```

Structure `Node` defines a tree node; this tree is used in algorithms a2, a4 and a5. Methods `FirstLeaf()`, `NextLeaf()`, `NextVertex()`, `BypassVertext()` and `Bypass()` provide ability to walk through the tree. Method `String()` returns a string (on the alphabet {A, C, T, G}, created from the values on the path from this node to the root; similar method `Values()` returns a vector of values (0 – 3) from each node on the path from this node to the root.

```
int getmindistance(std::string& lmer, std::string& strand,
unsigned& pos);
```

Function `getmindistance()` returns smallest hamming distance of the motif `lmer` on the string `strand` and stores its position on the string into the `pos` variable.

```
int gettotaldistance(std::string& lmer,
std::vector<std::string>& dna, std::vector<unsigned>& s);
```

Function `gettotaldistance()` returns smallest hamming distance of the motif `lmer` on the vector of strings `dna` and stores all positions into the vector `s`.

```
std::vector<unsigned> median_string_find_tree_bf(int length,
std::vector<std::string>& dna);
```

Function `median_string_find_tree_bf()` returns the vector of positions with a motif of length `length` which is common for all sequences stored in the vector `dna`. This function performs median search using tree and is not optimized.

```
std::vector<unsigned> median_string_find_tree_bound(int length,
std::vector<std::string>& dna);
```

Function `median_string_find_tree_bound()` returns the vector of positions with a motif of length `length` which is common for all sequences stored in the vector `dna`. This

function performs median search using tree and is optimized by branch-and-bound optimization.

```
std::vector<unsigned> find_motif_bf(int length,
std::vector<std::string>& dna);
```

Function `find_motif_bf()` returns the vector of positions with a motif of length `length` which is common for all sequences stored in the vector `dna`. This function performs median search using recursion and is not optimized.

```
std::vector<unsigned> find_motif_tree_bf(int length,
std::vector<std::string>& dna);
```

Function `find_motif_tree_bf()` returns the vector of positions with a motif of length `length` which is common for all sequences stored in the vector `dna`. This function performs median search using tree and is not optimized.

A.3 Reference of module lcs.cpp

```
void backtrace_single(std::string& string1, std::string&
string2, int **matrix, int rows, int cols, std::string& lcs);
```

Function `backtrace_single()` returns one LCS of strings `string1` and `string2`, using the matrix `matrix`, using backtracking from row `rows` and column `cols` to the left top corner of the matrix.

```
std::set<std::string> backtrace_all(std::string& string1,
std::string& string2, int **matrix, int rows, int cols);
```

Function `backtrace_all()` returns all LCS of strings `string1` and `string2`, using the matrix `matrix`, using backtracking from row `rows` and column `cols` to the left top corner of the matrix. This function is defined recursively.

```
void lcs(std::string& string1, std::string& string2);
```

Function `lcs()` the executive function that generates scoring matrix.

A.4 Reference of module align.cpp

```
void align(std::string& string1, std::string& string2);
```

Function `align()` calculates and displays local or global alignment of strings `string1` and `string2`. If string `string1` is shorter than 18 characters, used scoring matrix is also displayed. This method is influenced by two global variables – variable `gap` contains linear gap value and boolean variable `local` specifies type of alignment to perform.

A.5 Reference of module utils.cpp

```
std::string generate(int len);
```

Function `generate()` generates random string of specified length `len` over the alphabet {A, C, T, G}.

```
std::string readstring(char *filename);
```

Function `readstring()` reads the string from specified file and verifies its validity. If the file can't be opened or the string contains illegal characters, the standard C++ exception is thrown.

```
int c2n(char c);
```

Function `c2n()` translates character `c` from alphabet {A, C, G, T} to index 0 – 3.

```
char n2c(int n);
```

Function `n2c()` translates index `n` from 0 – 3 to the character of alphabet {A, C, G, T}.

```
int max3(int a, int b, int c);
```

Function `max3()` returns the greatest of three values